Concurrent Programming

Session 10: Cilk++

Computer Engineering Department
Iran University of Science and Technology
Tehran, Iran

Lecturer: Nima Ghaemian
Distributed Systems Lab.
Computer Engineering Department,
Iran University of Science and Technology,
nima@comp.iust.ac.ir

cilk_for Limitations

- Exactly one loop control variable
  - The loop initialization clause must assign the value.
- The loop control variable must not be modified in the loop body
  - WRONG: cilk_for (unsigned int i = 1; i < 16; ++i) i = f();
- The termination and increment values are evaluated once before starting the loop and
  - will not be re-evaluated at each iteration.
    - modifying either value within the loop body will not add or remove iterations
  - WRONG: cilk_for (unsigned int i = 1; i < x; ++i) x = f();
- The control variable must be declared in the loop header,
  - not outside the loop.
  - WRONG: int i; cilk_for (i = 0; i < 100; i++)

cilk_for Limitations (Cont'd)

- A *break* or *return* statement will *NOT work within the body of a cilk_for loop*
  - *the compiler will generate an error message*
- A *goto* can only be used within the body of a cilk_for loop if the target is within the loop body
  - WRONG CASES:
  - there is a *goto* transfer into or out of a cilk_for loop body
  - *goto* jumps into the body of a cilk_for loop from outside the loop

Cilk++ Concepts

- Strand
  - A concurrent agent consisting of a serial chain of instructions without any parallel control (such as a spawn, sync, return from a spawn, etc.).
- Concurrent Agent
  - A processor, process, thread, strand, or other entity that executes a program instruction sequence in a computing environment containing other such entities.
- Knot
  - A point at which the end of one strand meets the end of another. If a knot has one incoming strand and one outgoing strand, it is a *serial knot*. If it has one incoming strand and two outgoing strands, it is a *spawn knot*. If it has multiple incoming strands and one outgoing strand, it is a *sync knot*. A Cilk++ execution does not produce serial knots or knots with both multiple incoming and multiple outgoing strands.

Cilk++ Concepts (cont.)

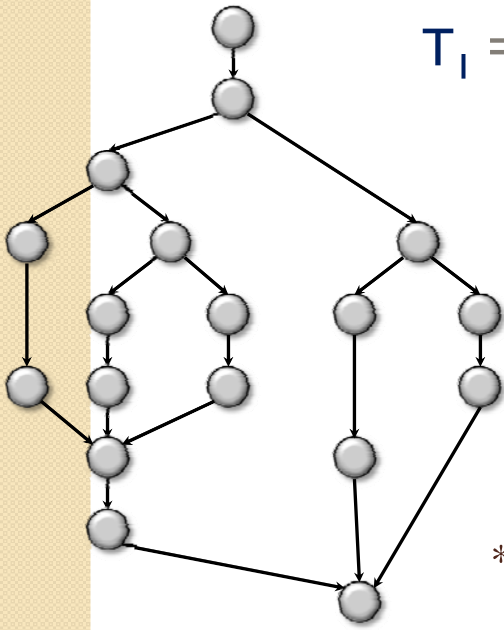
- Serial Consistency
  - The memory model for concurrency wherein the effect of concurrent agents is as if their operations on shared memory were interleaved in a global order consistent with the orders in which each agent executed them.
- Scale Down
  - The ability of a parallel application to run efficiently on one or a small number of processors.
- Scale Up
  - The ability of a parallel application to run efficiently on a large number of processors. See also linear speedup.
- Scale Out
  - The ability to run multiple copies of an application efficiently on a large number of processors.

Cilk++ Concepts (cont.)

- Span
  - The theoretically fastest execution time for a parallel program when run on an infinite number of processors, discounting overheads for communication and scheduling. Often denoted by T_∞ in the literature, and sometimes called critical-path length.
- Worker
  - A thread that, together with other workers, implements the Cilk++ runtime system's work stealing scheduler.

Work and Span

T_P = execution time on P processors



T_1 = ***work*** $\quad$ T_∞ = ***span***

Speedup on p processors

- T_1/T_P

Parallelism

- T_1/T_∞

* Also called ***critical-path length*** or ***computational depth***.

Cilk++ Concepts

- ***Parallelism*** is defined as the ratio of work to span, or T_1/T_∞.
- There are several ways to understand it:
  - The parallelism T_1/T_∞ is the average amount of work along each step of the critical path.
  - The parallelism T_1/T_∞ is the maximum possible speedup that can be obtained by any number of processors.
  - Perfect linear speedup cannot be obtained for any number of processors greater than the parallelism T_1/T_∞.

Sorting

- Sorting is possibly the most frequently executed operation in computing!
- **QuickSort** is the fastest sorting algorithm in practice with an average running time of O(N log N), (but $O(N^2)$ worst case performance)

Parallelizing Quicksort

- Serial Quicksort sorts an array S as follows:
  - If the number of elements in S is 0 or 1, then return.
  - Pick any element *v* in S. Call this pivot.
  - Partition the set S-{v} into two disjoint groups:
    - $\mathbf{S_1 = \{x \, \varepsilon \, S\text{-}\{v\} \mid x \leq v\}}$
    - $\mathbf{S_2 = \{x \, \varepsilon \, S\text{-}\{v\} \mid x \geq v\}}$
  - **Return quicksort(S_1) followed by v followed by quicksort(S_2)**

Parallel Quicksort (Basic)

- The second recursive call to *qsort* does not depend on the results of the first recursive call
- We have an opportunity to speed up the call by making both calls in parallel.

```
template <typename T>
void qsort(T begin, T end) {
  if (begin != end) {
     T middle = partition(
                   begin,
                   end,
                   bind2nd( less<typename
   iterator_traits<T>::value_type>(),
                            *begin )
                 );
     cilk_spawn qsort(begin, middle);
     qsort(max(begin + 1, middle), end);
     cilk_sync;
  }
}
```

Performance

- ./qsort 500000 -cilk_set_worker_count 1
  >> 0.122 seconds
- ./qsort 500000 -cilk_set_worker_count 4
  >> 0.034 seconds
- Speedup = T_1/T_4 = 0.122/0.034 = **3.58**

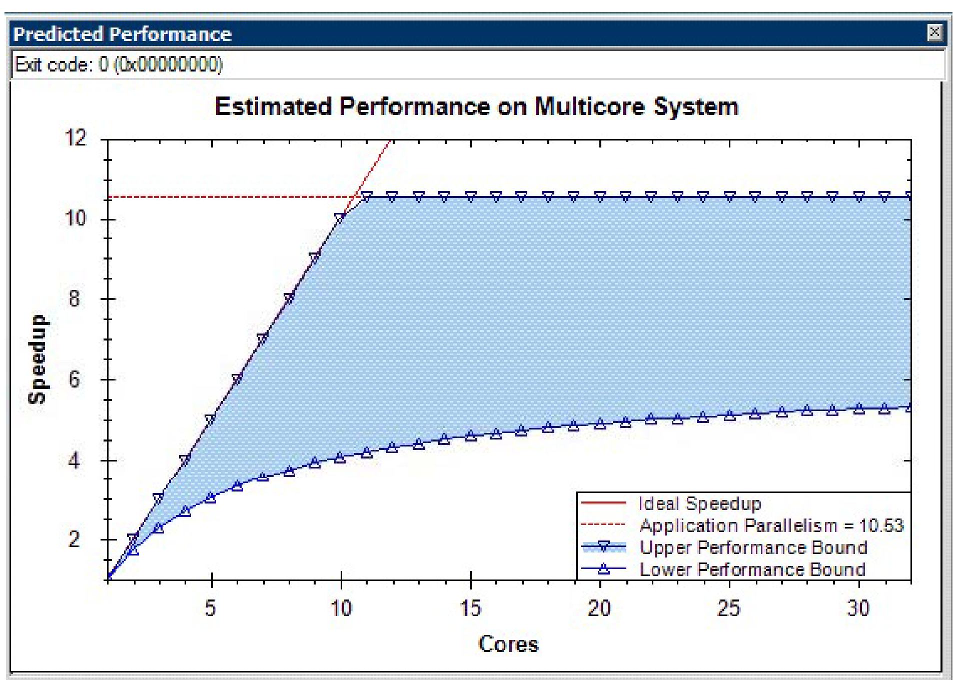
- ./qsort 50000000 -cilk_set_worker_count 1
  >> 14.54 seconds
- ./qsort 50000000 -cilk_set_worker_count 4
  >> 3.84 seconds
- Speedup = T_1/T_4 = 14.54/3.84 = **3.78**

Measure Work/Span Empirically

- cilkscreen -w ./qsort 50000000
  >> Sorting 50000000 integers
  >> work: 29696503161 instructions
  >> span: 5828326145 instructions
  >> parallelism: **5.1**

- cilkscreen -w ./qsort 500000
  >> Sorting 500000 integers
  >> work: 261593480 instructions
  >> span: 62178133 instructions
  >> parallelism: **4.2**

Parallel Performance Analyzer



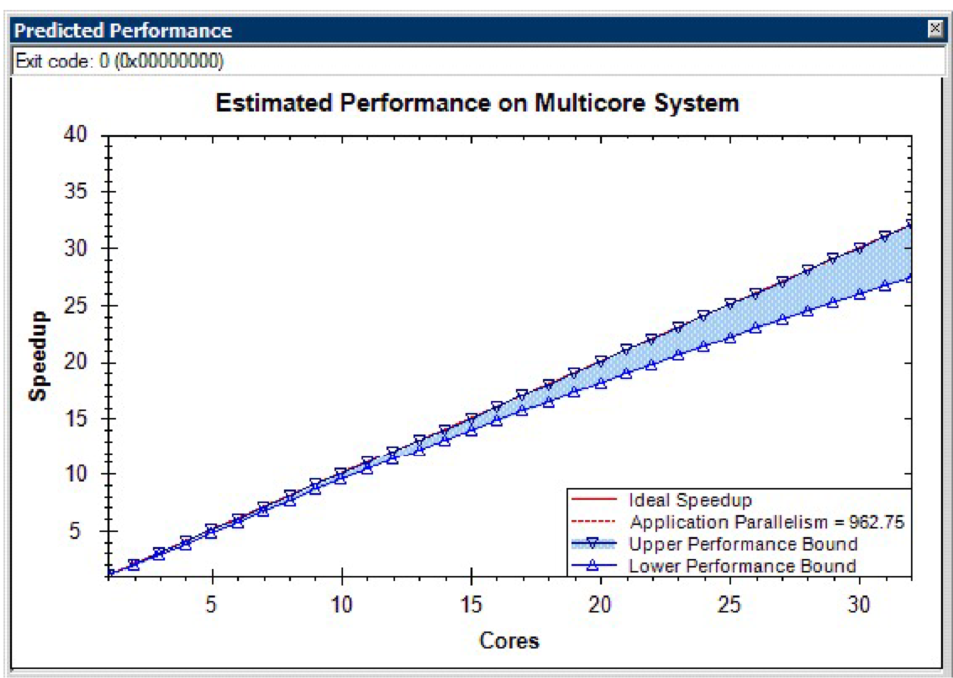
Performance Analysing

```
#define OUTER 1000
#define INNER 10000
#define ASIZE (INNER * OUTER)
double results[ASIZE];                    // array of 10,000,000 elements
// Initialize an array to values that depends on the array index.
// Assume Work(x) is an inexpensive function.
//
void SetArray(int x)
{
        results[x] = Work(x);
}

// Initialize an array to values that depends on the array index.
// Assume Work(x) is an inexpensive function.
//
void InnerLoop(int i)
{
        for (int j=0; j<INNER; ++j)
                SetArray((i * INNER) + j);
}


void OuterLoop()
{
        for (int i=0; i<OUTER; ++i)
                cilk_spawn InnerLoop(i);        // spawn the inner loop 1000 times
}
```

Parallel Performance Analyzer

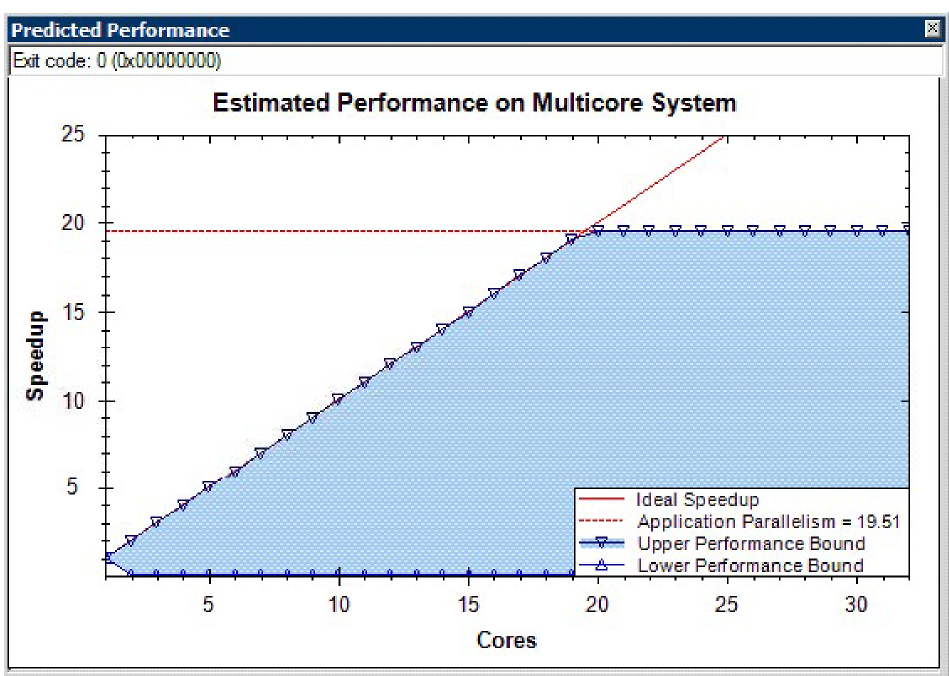


Improvement

```
void InnerLoop(int i)
{
        for (int j=0; j<INNER; ++j)
                cilk_spawn SetArray((i * INNER) + j);
}


void OuterLoop()
{
        for (int i=0; i<OUTER; ++i)
                cilk_spawn InnerLoop(i);
}
```

Parallel Performance Analyzer



Race Conditions

- Non-Local Variables
  - *"Untold confusion can result when the consequences of executing a procedure cannot be determined at the site of the procedure call"* - Wulf and Shaw
- passing the variable as an argument to function calls
  - Parameter Proliferation

Race Conditions (Cont'd)

- A Particular Matter…
  - If I have a code operating on a large data structure, why should I have to pass the data structure to each and every function that operates on the data structure?

Race Conditions (Cont'd)

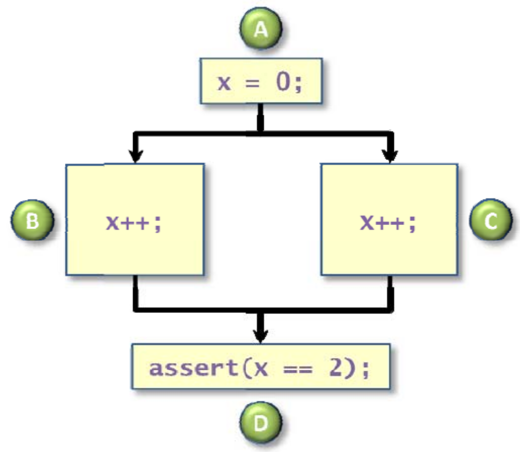
- Different types of race conditions
  - Depending on the synchronization methodology
  - Locking, condition variables etc.
- Determinacy Race
  - Deterministic
  - Nondeterministic

Race Conditions (Cont'd)

```
void incr (int *counter)
{
   *counter++;
}

void main() {
   int x=0;
   cilk_spawn incr (&x);
   incr (&x);
   cilk_sync;
   assert (x == 2);
}
```

DAG View



DAG View: non-atomicity